Names and Scopes in JML

A.Weigl

April 10, 2023

Introduction

The Java Language Specification (JLS, Chapter 6) defines named entities which can be referenced by name, and scopes which are code areas in which these names are valid. Additional to the Java entities, JML adds its own entities, and also, within JML we address JML and Java entities.

> A declared entity (§6.1) is a package, class type (normal or enum), interface type (normal or annotation type), member (class, interface, field, or method) of a reference type, type parameter (of a class, interface, method or constructor), parameter (to a method, constructor, or exception handler), or local variable. – JLS, Chapter 6

The name resolution describes how we can resolve a name to its refereeing entity.

In this remark, we want to clarify the named entities in JML, the scopes within JML, and the name resolution.

In the following, the use the notions "*in JML*" and "*in Java*" to specify the location of entities. An entity, e.g., a field declaration, is in *in JML*, if it is defined inside a JML annotation comment. "*In Java*" means the same and refers to the locations outside JML comments.

Basic Rules

Rule 1 *Within Java, we can not refer to any named entity in JML.*

JML lives inside comments, and is not evaluated or interpreted by any Java tool (compiler, javadoc, etc.), hence, it is not allowed to reference to a JML declared entity inside the regular Java code. On the other side, within JML we can refer to Java entities:

Rule 2 *A JML annotation comment inherits the Java scope in which it is written.*

Rule 3 handles the scope of JML statements inside constructors or method bodies, or the class-level annotation (e.g., invariants, axioms, etc.). The Rule 3

has exceptions, e.g., methods contracts can also refer to the parameters of the associated method. Further adjustments need to be made.

The name resolution should be unambiguous:

Rule 3 *Any name resolution conflict between a JML and Java entity is an error.*

If a name inside JML refers to a JML entity, but there exists also a Java entity with the same name in this scope, we consider this as an user error, and JML tools should decline further processing. As an example for an error, consider a forall-clause `forall int x;`, which hides a Java field declaration:

```
class A {
   public int x;

   //@ ensures (\forall int x; 0 <= x < 5; x >= 0); //! forbidden
   public void foo() {
     //@ ghost int x = 0; //! not a problem
     ...
     //@ set x = 1; //! problem: x is ambiguous.
   }
}
```

As a consequence, we are not able to shadow a Java variable inside JML, but we are able to shadow JML variables.

```
class A {
   public void foo() {
     //@ ghost int x = 0; //! not a problem
     ...
     //@ assert x != (\let x = 2; x);
   }
}
```

JML entities

Abstractedly spoken, inside JML comments we declare entities (fields, methods, invariants etc.) in the JML language. Some declarations can be referenced by name. The name resolution is the process which translates a given name, either a local or qualified identifier, into the entity by considering the current scope.

We define which entities are referenceable at all:

Rule 4 *Following JML named referenceable entities exists:*

- *declaration of model methods*
- *declaration of model and ghost fields*

- *model classes*
- *contract-local variable declaration by* `forall` *clauses.*
- *body-local variable declarations in ghost statements*
- *expression local variable declarations by binding expressions or* `signals` *clause.*

Note, that a class given in a separated JML file does not declare the class. Therefore, every external specified class (in a JML file) requires a counter-part declared in Java. This does not apply for model classes defined in JML files. Model classes are entities by themselves.

There exists JML constructs with a name, which are not (referenceable) entities. As a consequence we are not able to refer to them.

Rule 5 *The following constructs are not referable entities:*

- *method contracts*
- *contract clauses*
- *class-level specification clauses, e.g.,* `invariant`, `initially`, `represents`
- *labeled expressions (*`\old`*)*

Rule 5 is only valid for JML specification language. Your verification tool or proof language can use the names to reference to these constructs, but we do not allow this in JML. Hence, we also have no restriction on these name.

Rule 6 *Declaration in the same scope are not allowed to clash by their names.*

For example, the declarations clashes with their names.

```
class A {
  //! invalid: name clash
  //@ model void foo() {}
  //@ model int foo() {}
}

class A {
  //! invalid: name clash
  //@ invariant (\forall int x, int x; true);
}

class A {
  //@ forall int x;
  //@ forall int x; //! invalid: name clash
  void foo() {
    //@ ghost int x;
```

```
    {
        //@ ghost int x; //! absolute fine. shadowing of outer variable
    }
    ...
    //@ ghost int x; //! invalid: name clash
  }
}
```

Whereas the following is allowed, as invariants are not referenceable within JML. Nonetheless, Your tool might reject this.

```
class A {
  //@ invariant abc: true;
  //@ invariant abc: false;
}
```

Scopes

Rule 7 *Every JML-encapsulated Java construct behaves similarly regarding to name resolution.*

The name resolution in model classes or methods behaves identical to the classes or methods. The name resolution just extends to valid JML backslash names. Additionally, following JML contracts open a new scope:

- Binder expression
- signals-clause
- JML contracts

A scope is a semantical object, that defines a code area in which local references are resolved. Scopes are nested. A inner scope inherits the name from the outer scope. For example, the member variables of a class are accessible within the methods. New variables can be declared which shadowed the identically-named (Java or JML) variables from the outer scope.

Rule 8 (Binder expressions) *In a binder expressions, the bounded variables can only be referenced in the given expressions. Each occurrence of a bounded identifier refers to the bounded variable declaration in the closest-parent binder expression.*

Currently, the following binder expression are known:

- `let`-expression
- quantifier (`\forall`, `\forall`), Hilbert operator (`\choose`), `max`, `min`, `numof`.

All binder expression declare one or more parameters, which can be referenced inside the other arguments of the expression. Following expression are valid:

```
//@ ghost int x;
//@ assert (\forall int x; 0 <= x < 100; true) &&
           (\let int x = 5; x)                 ;
```

As a binder expression opens a new scope, its parameter shadows variables in the outer scope. Hence, in the example above there is no name clash.

Rule 9 (Signals-clause) *A signals-clause* `signals (T e) <expr>;` *declares a new scope, in which* `e` *is bound to a non-null instance of type* `T`*. The scope only spans over the given expression.*

The scope definition of JML contracts are more difficult. We have to deal with the method parameters, return value, nested contracts, and also forall-clause.

Rule 10 (JML contracts) *In the scope of a JML contract, the following names are defined and shadowed in the given order:*

1. *Names inherited from the parent scope.*
2. *For method contracts, the method parameters.*
3. *For method contracts of non-void methods, variable* `\result` *is bound.*
4. *Every variable declared in a forall-clause.*

The inheritance (Item 1) in has a different meaning for different contracts. For method contract, the parent scope is the class scope and hence consists the member variables, static variables, etc. For local contracts (block contract, loop contract) the parent scope is the scope of the block statement containing the local contract. Therefore, local variables which are declare previously on this block statement are available additionally to the class scope. Sub-contracts inherits the scope from the parent contract.

For loop invariant and contracts, the access to the loop variable of `for`-loops must be possible, which is currently not covered by Rule .

Rule 11 (Loop variable) *The declared loop variables of* `for` *and* `for-each` *loops are declared in the loop invariant or contract.*